Investigating the Approximation Power of Neural Networks

(Final Report)

Submitted April 26, 2021, in partial fulfillment of
the conditions for the award of the degree **BSc Computer Science.**

Hongyan Deng

20030435

Supervised by Amin Farjudian

School of Computer Science University of Nottingham

Abstract

Neural networks provide a computational model for solving various types of problems. They are able to solve complex equations, such as differential equations, therefore, they could be widely applied in various field, for instance, computational fluid dynamics. Nevertheless, for traditional neural network methods, an unique network is required for a different equation and its slight modification, which is inefficient and inconvenient. Besides, neural networks are also suitable for approximating unknown functions with less model parameters required compared with traditional finite element method. Based on the theoretical research, the proof has been deduced that any function could be approximated with any desired accuracy. Plenty of experiments regarding to various types of network activation functions are implemented to gain approximation results to verify the theoretical arguments. However, the arguments related to ReLU network have not been verified yet.

Therefore, this project aims to deal with two parts: 1) investigate the computational power of neural networks by solving parametric ordinary differential equations with sigmoid activation function 2) investigate the approximation ability of neural networks by performing experiments to verify the theoretical arguments of ReLU network.

For the first part, satisfying results are acquired through experiments. For the second part, we emphasize the gap between experimental results and the theoretical arguments.

Acknowledgement

I would like to deliver my sincere appreciation for Dr. Amin Farjudian who was very patient in providing me guidance during the project and supported me to dig further in this field. He also offered me lots of significant suggestions when I encountered difficulties. Hope everything will be well in the future for him and UNNC.

Contents

List of Tables

List of Figures

Acronyms

ANN	Artificial Neural Network
CFD	Computational Fluid Dynamic
FEM	Finite Element Method
GDP	Gross Domestic Product
MLP	Multi-layered Perceptron
MLPNN	Multi-layered Perceptron Neural Network
NN	Neural Network
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation

Chapter 1

Introduction

Neural networks provide a computational model for solving various types of problems. It is acknowledged to us that neural networks (NNs) have achieved huge accomplishments at pattern recognition tasks such as facial identification and object recognition. Apart from those prevalent applications, they also could be used to perform symbolic reasoning by solving mathematical expressions. Solving complex equations, such as differential equations, is worthwhile to investigate as they have various engineering applications in computational fluid dynamics (CFDs), signal processing, electromagnetics, etc. One of the strong examples of applying NNs to solve mathematical expressions in real life is Wolfram Alpha, a famous search engine, which is capable of searching and solving math equations developed by using Wolfram language to integrate, train and deploy NNs systems.

The very fact that NNs are suitable for approximating unknown functions is a consequence of classical results in foundations of machine learning, whereby it has been proven that for some important function spaces, any function may be approximated to any required degree of accuracy using neural networks. Approximation results are generally classified according to the activation function of the network. For sigmoid activation function, the theoretical results have been verified through experiments. For ReLU networks, however, the theoretical results have not yet been verified in all cases by experiments.

Traditional methods of solving differential equations require designing a network for each equation. Any slight modification of the equation requires designing a new network. There are, however, problems in practical, such as fluid dynamics, where a parametrized set of equations must be solved. Therefore, the aim of this project is two-fold: we first investigate the computational power of neural networks with sigmoidal activation functions for solving parametric ordinary differential equations (ODEs). We design a trial solution and a NN based on Lagaris's methods [1], which can solve an example ODE with parametric initial value via minimizing the cost function. In the second part, we perform some experiments to verify the theoretical results [2] regarding the approximation power of ReLU networks. We train a ReLU network to approximate a test function with two dimensional inputs. In particular, we highlight the gap between experimental results and the bounds provided in the literature.

1.1 Background

1.1.1 Background of Solving Parametric ODEs

There are lots of numerical strategies for solving differential equations subject to initial or boundary conditions. Among them, finite difference, finite element, finite volume, Runge–Kutta, boundary element are practical and prevalent methods involving the field of applied mathematics [3]. For the finite difference, finite element methods, the approximative solutions could be worked out at particularly preassigned grids by using the numerical operators [4]. Runge–Kutta methods could be applied to deal with ODEs on preassigned grid points [5]. However, generally speaking, the major drawback of those methods is computational complexity as they require lots of model parameters [6].

Therefore, lots of NN methods are proposed to solve difference equations more efficiently, for instance, parallel algorithm with Hopfield neural network [7], feed forward NN methods to solve linear and nonlinear ODEs [8], etc. Applying NN methods requires to adjust parameters and minimize the cost function within each iteration of training process. The essence of adopting NN methods is to construct appropriate trial solutions to satisfy the initial or boundary conditions and reduce the relative errors between the analytic solutions and the result solutions.

The reason why plentiful techniques are developed is that it is worthwhile to concern about the solutions of differential equations as they could be applied to various engineering disciplines [6], for instance, CFDs, signal processing, electromagnetics, etc. Those areas require precise and massive computation, and some issues with respect to them could puzzle for century, such as Navier-Stokes equations. Apart from those applications, they also could be applied to the fields of finance and economics. For example, ODEs could be applied to deterministic systems to work out gross domestic product (GDP) model, stochastic systems to derive the expected discounted penalty, and so on [9].

Nevertheless, the traditional NN methods of solving differential equations are constrained by a unique NN for each equation. A slight modification of the equation, such as changing the boundary values, enforces to train another new one. Moreover, some issues related to CFDs require to solve some parametrized set of equations. Those equations includes field parametrization and boundary values parametrization. However, changing the fields or the boundary values of the input parameters are not necessary to retrain a new NN, which could be less efficient and less convenient. Therefore, it is significant to adopt NN techniques to solve parametric differential equations more effectively. Due to the difficulties of parametrizing fields of a function, we start from dealing with parametrizing boundary values of ODEs.

1.1.2 Background of NN Approximation Power

Apart from the ability of solving differential equations with higher accuracy and less time consuming, the approximation capability for unknown functions is also worth to investigate. Except solving differential equations numerically, finite element method (FEM) can be applied to approximate unknown functions considered as a traditional approximation method with flexibility and feasibility. However, compared with FEM, NN methods to approximate functions are more powerful with fewer model parameters required proved by [2] and concluded in Section 2.4.2. Thus, in light of the fair approximation results of any unknown functions, the ultimate approximation capabilities of feedforward networks gain lots of attentions in research area.

Kolmogorov propose superposition theorem which demonstrates the approximation ability of multilayer feedforward networks [10]. NNs with sigmoidal activation function are used to approximate unknown mapping whose approximation accuracy could be improved with the increase number of hidden units [11]. Then, Cybenko [12] justifies that arbitrary feedforward networks can be considered as universal approximators. Nevertheless, his results are limited to sigmoidal activation function, in 1989, Stinchcombe and White demonstrate single-layer feedforward networks with arbitrary squash functions can work as universal approximators. Not only the unknown mapping can be approximated, but also their derivatives are able to be approached. Hornik et al. [13] provide the conditions to prove that multilayer feedforward networks with single hidden layer and a smooth activation function can approach any function derivatives.

The powerful approximation ability supports the usage of feedforward networks to be applied to any field requiring the approximation of unknown functions and their derivatives. For instance, it could be applied to economic area by offering aids to justify the correctness of a hypothesis. The results provided by Hornik et al. [13] aid Vinod and Ullah [14] to establish NN models as alternative framework to study the theory of firm and consumer. A realistic CFD design problem applying three-dimensions Navier-Stokes simulations can be solved by the hybrid approach of NN approximation proposed by Poloni et al. [15]. One of the technology described by them to address sensitivity derivatives calculation issues is NN method to approximate derivatives. Since a NN could be considered as a smooth approximator to approach multi-dimensional functions, it can be applied to approximate the sensitivity problems. Therefore, NN methods are powerful techniques to approach function with less computation costs and higher approximation accuracy.

1.2 Motivation

It is practical only for simple geometries to apply analytical solution methods that constrains the applicability [6]. Applying FEMs to solve them numerically could help to deal with practical problems with complex boundary conditions, however, it has major disadvantage regarding to computational complexity [6]. Some of them fail to be worked out in reality as they are only feasible theoretically. Therefore, applying neural networks to solve differential equations might be an alternative approach to traditional methods obtaining high computation efficiency.

In the field of uncertainty quantifications, they often involve physical and engineering applications which contain differential equations with random coefficient field [16]. Whereas, it could be tedious to train a new NN each time when solving the same type of differential equation with distinguished field values or boundary values. To address those issues, it appears to become necessary to train NNs to deal with parametric differential equations. Additionally, adopting NN methods to parameterize the quantity as function of input coefficients could help to reduce the curse of dimensionality in numerical differential equations [16]. Hence, it is worthy of investigating solving parametric differential equations based on neural network methods.

Apart from the ability of solving various types of mathematical functions by providing computational models, NNs are also suitable for approaching unknown mapping. By virtue of the proofs that any unknown function could be approximated to a satisfying accuracy by adopting NNs, massive novel techniques are developed based on this techniques. Considerable researches are conducted to investigate how the different activation functions affect the approximation results. For instance, plenty of experiments have already verified the theoretical results of sigmoid activation function. Whereas the theoretical arguments for ReLU network have not been verified yet. The second part of this project attempts to design experiments to verify the theoretical results of the approximation ability of ReLU network.

1.3 Aims and Objectives

The main object of this project is divided into two parts: Firstly, we apply NN methods to solve ODEs with parametric boundary values. The main reference for the first part is the method originally developed in [1]. Our main aim is to design a method for solving some classes of parametric ODEs using neural networks, by extending the methodology of [1].

Secondly, we would like to investigate the approximation power of NNs. Based on the theoretical arguments concluded by [2], we aim at performing experiments to verify the proof of ReLU network.

The Key Objectives of The Research:

1. Investigate previous methods of solving different types of ODE based on NN methods.

2. Design and optimize methods of solving a certain class of ***parametric*** ODEs by using neural networks.

3. Implement and train the designed neural networks.
4. Evaluate the performance of the networks.
5. Investigate the approximation power of NNs
6. Perform experiments to verify the theoretical arguments of ***ReLU*** network.

1.4 Contributions

1. Propose a trial solution and design a NN to solve a test ODE with ***parametric initial values*** in Section 3.1
2. Evaluate the solution of the implemented NN in Section 4.1.
3. Propose an alternative ***geometric*** proof for ReLU network in Section 3.2.
4. Conduct experiments to verify the theoretical argument regarding the approximation power of ReLU network. By using ***Sobolve norm*** to evaluate the approximation accuracy of a function and its derivatives in Section 4.2.1.
5. Conduct experiments to verify the theoretical argument regarding the approximation power of ReLU network. By using ***L2-norm*** to evaluate the approximation accuracy of a function in Section 4.2.2.

Chapter 2

Related Works

2.1 Artificial Neural Networks

A computation model of neural networks inspired by biological NNs was brought about by McCulloch and Pitts spurring up the development of artificial neural networks (ANNs), which mimic the learning process of the human brain to extract features or patterns from input data and could be applied to various disciplines and fields [17][18]. A perceptron is a parallel computer that contains many readers and can independently scan a field at the same time [19]. For simple perceptron, a function or a calculation model is required to generate the output. For multi-layered perceptron, there are at least three layers with one or more hidden layers [20]. They generate nonlinear mapping from the input layer to the output layer, which could be applied to function approximation with feasible computation [21]. Thus, feedforward multi-layered perceptron (MLP) neural networks could be applied to produce approximation solution of differential equations from the optimum adjustable parameters, weights and biases, through minimizing the cost function. Besides, one of the experiences related to promoting the accuracy of training a MLP should be noticed that the number of neurons in the hidden layer is critical instead of the number of hidden layers [22]. Fig. 2.1 shows the mathematical model of an ANN.

As the figure shown, a perceptron accepts n inputs and for each input has a weight connecting between the neuron and itself. And there is a bise assigned for each neuron in the hidden layer aiming at adjusting the mapping function to improve accuracy. Then the output of the perceptron will be passed into an activation function to produce an output, whose types could have linear, sign, sigmoid and step functions [23].

2.2 Neural Network Methods for Solving Differential Equations

Since using the most general methods, finite difference, finite elements to solve differential equations could increase the computation burden rapidly by the increased discretization

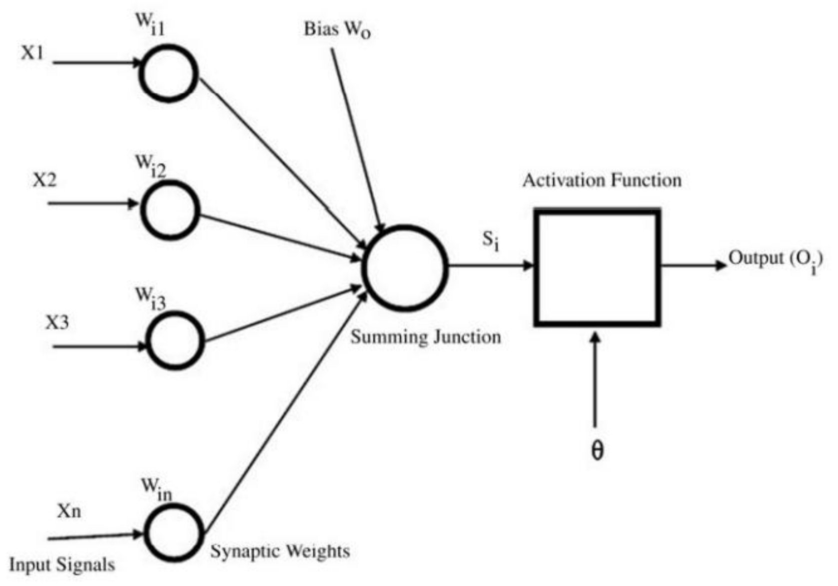


Figure 2.1: Mathematical model of artificial neural network

points, plenty of NN methods are presented to solve complex finite difference equations [3]. Parallel algorithm with Hopfield neural network models is proposed to solve first order differential equations [7]. Linear and nonlinear ODEs could be solved by applying feed forward NN methods presented by Meade and Fernandez in 1994 [8]. After that, hybrid numerical methods are brought forward based on NN methods and optimization techniques aiming at working out high order differential equations with approximated solutions [24][3]. Furthermore, multi-layered perceptron neural network (MLPNN) method could be used to solve a class of differential equations based on recent development, such as applying MLP to deal with fuzzy differential equations and vibration control problem. The most prominent advantage of applying NN methods to deal with differential equations is generating differentiable solutions in closed analytic forms compared to other techniques providing discrete solutions with constrained differentiability. Moreover, they obtain fair generalization properties without heavy computations when the number of discretization points is increased [23].

Apart from dealing with exact differential equations, solving differential equations with parametric initial conditions or fields is also worth of researching. It should be noticed that parametric ODEs have not been dealt with in the literature related to handling standard ODEs and they are non-classical [25]. In 2015, the general solutions for three classes of first- and second-order nonlinear ODEs are proposed by Polyanina and Zhurov, which deal with parametrically defined ODEs and they could be further applied to generate exact solutions [26]. NN methods also could be applied in solving differential equations parametrically due to their applicability in function approximation. A relatively novel approach was presented in 2010 for solving fuzzy differential equations by replacing them to equivalent parametric form and ODEs system. This method yields more accuracy compared with numerical methods, and even could gain better results with more neurons or training points [27].

Although the approximations could be fairly closed to analytical solutions and the computation burden could be addressed compared to traditional finite differential methods, dimensional curse for gradient computation also exists while solving high order differential equations. In 2019, Chen and Duvenaud constructed a NN that permits a family of differential operators to be calculated with reasonable dimension derivatives [28]. Recently, a novel proposed approach called neural ODEs were proposed, which parametrizes the hidden unit dynamics by millions of leaned parameters. It could be considered as a replacement for deep neural networks with memory efficiency and adaptive computation [29].

2.3 Neural Network Methods to Solve ODEs

The texts below will briefly summarize Lagaris's works regarding ODEs, which include the description of multilayer perceptron NN method, the gradient computation and illustration of Lagaris's method about solving single ODEs and systems of couples ODEs [1].

2.3.1 Multi-layer Perceptrons to Solve ODEs

As introduced before, MLPNN applies feedforward NN as the foundational approach to approximate the solution of a function. Assuming a discretization of the domain D and the boundary S into i points, which are represented as $\hat{D}$ and $\hat{S}$ respectively. Then the designed trial solution $\Psi_t(\vec{x}, \vec{p})$ employing a feedforward NN with parameters $\vec{p}$ required to be adjusted, which correspond to the weight and biases of the NN will be served as training equation to minimize the computational errors with optimization techniques, such as gradient decent. The minimization formular is:

$$min_{\vec{p}} \sum_{\vec{x}_i \in \hat{D}} G(\vec{x}_i, \Psi_t(\vec{x}_i, \vec{p}), \nabla\Psi(\vec{x}_i, \vec{p}), \nabla^2\Psi(\vec{x}_i, \vec{p}))^2. \tag{2.1}$$

For the designed trial solution $\Psi_t(\vec{x})$, it could be constructed as a sum of two terms satisfying the BCs.Through adjusting parameters in each iteration, the errors could be minimized after training. Such process could transform the problem from the constrained to the unconstrained optimization, which is considered as feasible to solve with less computation burden. Hence, proposing a practicable and justifiable trial solution for distinguished ODE problems with fewer errors is critical for this project.

2.3.2 Constructing Trial Solutions to Solve Different ODEs

Here three main methods are summarized as follow including the first and second order of the initial value problem, the second order boundary value problem and the system of kth order problem.

Initial Value Problem

Considering the first order ODEs with $x \in [\,0,\,1\,]$ and the initial condition $\Psi(0) = A$:

$$\frac{d\Psi(x)}{dx} = f(x, \Psi).$$

A designed trial solution is given as where $N(x,\, \vec{p})$ is the output of the network, $\Psi_t(x)$ satisfies the initial condition.:

$$\Psi_t(x) = A + xN(x, \vec{p}).$$

The cost function could be minimized as follow:

$$E[\vec{p}] = \sum_i \{\frac{d\Psi_t(x_i)}{dx} - f(x_i, \Psi_t(x_i))\}^2,$$

where, $\frac{d\Psi_t(x)}{dx} = N(x, \vec{p}) + x\frac{dN(x,\vec{p})}{dx}$.

Boundary Value Problem

For the two points Dirichlet problem, boundary conditions are $\Psi(0) = A$ and $\Psi(1) = B$, the trial solution could be formed as to satisfy the initial condition:

$$\Psi_t(x) = A(1 - x) + Bx + x(1 - x)N(x, \vec{p}).$$

System of K^{th} Order ODEs

As for the system of K^{th} order ODEs with $\Psi_i(0) = A_i$, $(i = 1, \ldots, K)$ as initial conditions:

$$\frac{d\Psi_i}{dx} = f_i(x, \Psi_1, \Psi_2, \ldots, \Psi_K).$$

Each trial solution should construct one feedforward NN. The cost function is given as follow:

$$E[\vec{p}] = \sum_{k=1}^{K} \sum_i \{\frac{d\Psi_{t_k}(x_i)}{dx} - f_k(x_i, \Psi_{t_1}, \Psi_{t_2}, \ldots, \Psi_{t_k})\}^2.$$

2.3.3 Gradient Computation

The process of minimizing the cost of NN as equation 2.1 is considered as the process of training NNs. Not only the output of the NN is significant for training, the derivatives of NN inputs are also important to minimize the cost function. Furthermore, in order to update the parameters, it is also necessary to calculate the derivatives of each parameter.

Gradient Computation with respect to Network Inputs

Consider a feedforward NN with n inputs, one hidden layer with H sigmoid nodes and a linear output. For a given input vector $\vec{x} = (x_1, ..., x_n)$ the output of the network is $N = \sum_{i=1}^{H} v_i \sigma(z_i)$ where $z_i = b_i + \sum_{j=1}^{n} w_{ij} x_j$. It is straightforward to show the derivatives with respect to its inputs

$$\frac{\partial^k N}{\partial x_j^k} = \sum_{i=1}^{H} v_i w_{ij}^k \sigma^{(k)}(z_i),$$

where and $\sigma(k)$ denotes the k_{th} order derivative of the sigmoid. Moreover it could be rewrote as

$$\frac{\partial^{\lambda_1}}{\partial x_1^{\lambda_1}} \frac{\partial^{\lambda_2}}{\partial x_2^{\lambda_2}} ... \frac{\partial^{\lambda_n}}{\partial x_n^{\lambda_n}} N = \sum_{i=1}^{n} v_i P_i \sigma^{(\Lambda)}(z_i),$$

where

$$P_i = \prod_{k=1}^{n} w_{ik}^{\lambda_k},$$

and $\Lambda = \sum_{i=1}^{n} \lambda_i$.

Gradient Computation with respect to Network Parameters

In another words, the derivatives with respect to NN inputs are equivalent to a feedforward neural network $N_g(\vec{x})$ with one hidden layer. The NN parameters are replaced with $v_i P_i$. Furthermore, the transfer function of each hidden node is replaced with the Λ^{th} order derivative of the sigmoid function. The derivatives of the NN parameters, w_{ij}, v_i, and b_i are expressed as

$$\begin{aligned} \frac{\partial N_g}{\partial v_i} &= P_I \sigma^{(\Lambda)}(z_i), \\ \frac{\partial N_g}{\partial b_i} &= v_i P_I \sigma^{(\Lambda+1)}(z_i), \\ \frac{\partial N_g}{\partial w_{ij}} &= v_i P_I \sigma^{(\Lambda+1)}(z_i) + v_i \lambda_j w_{ij}^{\lambda_j - 1} \Big(\prod_{k=1, k \neq j} w_{ik}^{\lambda_k} \Big) \sigma^{(\Lambda)}(z_i). \end{aligned} \tag{2.2}$$

2.4 Approximation Power of Neural Networks

2.4.1 Applying Multilayer Feedforward Networks to Approximate Function Solutions

In 1969, Minsky and Papert [19] first propose that the simple two-layer perceptron is able to approximate function solutions without restriction to specific class. Due to fair approximation results of any applicative functions, the ultimate approximation capabilities of feedforward networks gain lots of attentions in research. Kolmogorov's superposition theorem [10] [30] demonstrated the approximation capabilities of multilayer feedforward networks, however, a different transformation is required for each continuous function to be approximated, which constrains the intermediate units for representation.

After that, more specific squashing functions [31], such as sigmoid function and logistic function, are applied in the research of Lapedes and Farber [11], and Irie and Miyake [32], which could approximate function results by increasing the number of hidden nodes to improve accuracy without the limitation of function types. Nevertheless, their demonstrations are based on a particular class of feedforward networks with one or two hidden layers, which miss the justifications of arbitrary feedforward networks working as universal approximators [33]. Then, Cybenko [12] proposed that continuous feedforward networks with one hidden layer and any consecutive sigmoidal nonlinearity could approximate arbitrary decision region with satisfactory accuracy. On the account of the Hahn-Banach and Riesz Representation Theorems, he demonstrated that any collection of compact and disjoint subsets of R^n can be discriminated with arbitrary precision. Compared to other majority approximation methods which suffer from the curse of dimensionality and require astronomical numbers of terms investigated by Makhoul and Schwartz [34], his approximating properties are considered as powerful and efficient.

Their results are based on the application of sigmoid activation function. While, in 1989, Stinchcombe and White rigorously illustrate that multilayer feedforward networks with single hidden layer and arbitrary squashing functions could be considered as universal approximators, since they are able to approximate any Borel measurable function to a satisfactory accuracy by providing sufficient hidden nodes [33]. By applying the Stone-Weier-strass Theorem and the cosine squasher of Gallant and White [35], any function could be approximated by using arbitrary squashing functions. Hence, theoretical demonstration gives this conclusion, even though application failures could be generated due to inadequate training and number of hidden nodes.

In addition to approximating unknown mapping, it is also necessary to investigate how to approximate unknown derivatives. Therefore, Jordan conducted experiments to conclude the results related to robot learning of smooth movements that the key step of his approach is to learn adequate approximation to Jacobian matrix of an unknown mapping [36]. Although his experiments went successfully, they are still unable to guarantee multilayer feedforward networks obtain the ability to approximate arbitrary functions and their derivatives theoretically. Then, Hornik et al. [13] gave conditions to ensure multilayer feedforward networks could generate accurate approximation of an arbitrary function and its derivatives with a single hidden layer and a smooth activation function. They also investigated that these networks are capable of approximating non-differentiable functions in classical theories, but obtain general derivatives, such as certain piecewise differentiable functions. Furthermore, they justified the application of these networks, which requires simultaneous approximation of a function and its derivatives.

2.4.2 Comparing the Number of Parameters in Finite Element Method and ReLU Networks

The basic idea of gaining solutions via applying FEM is to find the local solutions within a sub-domain split from the computational domain and stich the individual solutions of those patches to group a global solution [37]. Finite element method could be applied to solve differential equations numerically regarding to its flexibility and feasibility of solving numerical problems. It is also a strong and flexible way of approximating functions. Compared with FEM, neural networks also could be considered as powerful method to approximate results as mentioned before. Feedforward neural networks with ReLU activation function are proposed as one of the NN methods as universal approximators. By comparing the number of model parameters for two methods, the approximation capability could be concluded. If the same approximation accuracy is reached, the one approximating the results with less parameters would be more efficient. Therefore, if we would like to compare the approximation ability of two methods and focus on the more efficient one, we need to compare their parameter number at first.

As for FEM, before approximating results, the discretization process should be performed. Considering two dimensions, the domain should be subdivided into multiple triangles as shown in figure. The solution of a function will be the approximation results given by vertices of each triangle. Between those vertices, the solution is linearly interpolated. Therefore, the number of parameters required by FEM is the number of vertices in total. The more triangles the domain divide into, the shaper changes each region would obtain, which means higher accuracy for the solution. Hence, the number of parameters could be expressed as

$$size_{FEM} = O(\epsilon^{-1}),\ with \|f - f_{FEM}\| \leq \epsilon,$$

where f is a real value function, f_{FEM} is the approximation result applying FEM and ε is the relative error for approximation.

With respect to NN method, the size of the NN could be conducted as a measurement of neural networks' convergence speed. Montanelli and Du [2] proposed, for general neural networks f_N and a real value function f in $\mathbb{R}^d$ with input dimension d and the order of integrable m, the size of the neural network satisfies:

$$size_{ReLU} = O(\epsilon^{-\frac{d}{m}}),\ with \|f - f_N\| \leq \epsilon. \tag{2.3}$$

They also listed the deduced results of shallow and deep networks with different activation functions based on the general form 2.3. The function spaces they considered are Sobolev spaces $W^{m,p}(\Omega)$ which have partial derivatives in $L^p(\Omega)$ (for some compact subset $\Omega \subset \mathbb{R}^d$) up to the order m. The formula is expressed as:

$$W^{m,p}(\Omega) = \left\{ f \in L^p(\Omega) : D^{\boldsymbol{k}} f \in L^p(\Omega), |\boldsymbol{k}|_1 \leq m \right\},$$

with multi-index $\boldsymbol{k} = (k_1, ..., k_d) \in \mathbb{N}^d, |\boldsymbol{k}|_1 = \sum_{j=1}^{d} k_j$, and partial derivatives

$$D^{\boldsymbol{k}} f = \frac{\partial^{|\boldsymbol{k}|_1} f}{\partial x_1^{k_1} ... \partial x_d^{k_d}}.$$

Concluded in the table, for shallow and deep NNs with ReLU activation function, their NN size could be expressed as below:

	Shallow	**Deep**
ReLU	$size = O(\epsilon^{-\frac{d}{m}})$, $f \in W^{m,2}(B^d)$	$size = O(\epsilon^{-\frac{d}{m}} \lvert log_2 \epsilon \rvert)$, $f \in W^{m,\infty}([0,1]^d)$

Table 2.1: Approximation Results

where the domain is either a hypercube, the unit ball B^d of $\mathbb{R}^d$ with respect to the two-norm of vectors.

Based on the formula above, to reach the same relative errors, the number of parameters required by two methods should satisfy the relation that $size_{FEM} \leq size_{ReLU}$. Although the approximation results of NNs suffer from the curse of dimensionality, it is justifiable to conclude that NN method with ReLU activation function is more efficient than FEM with fewer parameters required.

Chapter 3

Methodology

3.1 Proposed Method to Solve Parametric ODEs

The approach proposed by Lagaris et al. [1] is able to solve different types of ODEs or partial differential equations (PDEs) with relatively small relative errors by applying trial solutions to minimize the cost function. Their method, however, has been developed for functions with known initial values or boundary values. Thus, the trial solutions proposed by Lagaris are unable to generate results for functions with parametric initial conditions or field. We proposed a method attempting to solve parametric ODEs with parametric initial condition based on the methodologies of Lagaris [1].

3.1.1 Proposed Trial Solution

The proposed trial solution of parametric initial condition for ODEs obtains two input values: the initial value, $x_1 = a$, and the ODE parameter, $x_2 = x$, where $x \in [\,x_0,\, x_0 + T\,]$, $a \in [\,a_0,\, a_0 + S\,]$. It could be expressed as:

$$\Psi_t(a,\, x) = a + (x - x_0)N(a, x, \vec{p}), \tag{3.1}$$

where, the boundary term $A(\vec{x})$ containing no adjustable parameters in Lagaris methods should be replaced to the adjustable variable a discretized as inputs x in its domain $a \in [\,a_0,\, a_0 + S\,]$. The network is enforced by two discretized parameters x and a, which should be treated equally considered as inputs containing in an array. The reason why we enforce both of them in a network is that we suggest they both could influence the training process. For higher solution accuracy, we should enforce both of them. This is proved by experiment results with much higher accuracy and reasonable training process. As the relative errors measured for the network only containing x as parameters could obtain substantial increase during the training process, which is considered as abnormal, the network enforcing both two parameters are more sane with relatively satisfying decreasing trend. The term $(x - x_0)$ is for satisfying the initial condition.

In addition, it is worth to notice that since there is an initial value parameter also required to be discretized, the size of the weights, w_{ij}, from the input layer to the hidden units should be doubled for the column dimension.

3.1.2 Proposed Cost Function

The weights and biases are adjusted by applying optimization methods to minimize the cost function. The error quality is written as:

$$E[\vec{p}] = \sum_{i=1}^{M}\sum_{j=1}^{K}\{\frac{d\Psi_t(a_i, x_j)}{dx} - f(x_j, \Psi_t(a_i - x_j))\}^2, \tag{3.2}$$

where M is the number of discretization points for $a = \{a_1, \dots, a_M\}$, and K is the number of discretization points for parametric initial condition $x = \{x_1, \dots, x_K\}$. Thus, we could have $d\Psi_t/dx = N(a, x, \vec{p}) + (x - x_0)dN(a, x, \vec{p})/dx$. Note, the derivative of trial solution should only be with respect to inputs x. Then, it is straightforward to compute the derivatives of the errors with respect to the parameter $\vec{p}$, the weights and biases, based on the equations 2.2.

3.2 Alternative Geometric Proof for ReLU Networks

In this section, an alternative proof is proposed from geometric perspective that ReLU networks method is more efficient to approximate solutions than FEM. If the geometric partitioning is able to deliver smaller or equal number of parameters required by ReLU networks compared with FEM to reach the same relative errors, we could say prove founded. In order to deduce the proof, it is necessary to demonstrate the process of geometric partition for ReLU NN. Currently, we only illustrate 2 dimensions for input ($d = 2$). As for higher dimensions, the proof process should be the same.

As shown in the previous chapter, the NN with ReLU activation function could be expressed as

$$N = \sum_{j=1}^{n} v_j ReLU(\delta_j(x_1, \dots, x_d))$$

$$\delta_j(x_1, \dots, x_d) = \sum_{i=1}^{d} w_{ij}x_i + u_j$$

where d is the input dimensions, n is the number of hidden nodes. w_{ij} is the weight between input i and hidden node j, u_j is the bias for hidden node j, and v_j is the weight between hidden node j and the output.

If we set $\delta_j(x_1, \dots, x_d) = 0$, we could get a hyperplane for each hidden node. The arrangement of those hyperplanes could divide the domain region into arbitrary patches.

For each patch, it could be considered as a finite element. In other words, the hyperplanes partition the function domain into regions, where each region may be interpreted as a finite element. For instance, in two dimensions ($d = 2$), for hidden node j, we could get a line $w_{1j}x_1 + w_{2j}x_2 + u_j = 0$ in the plane $\mathbb{R}^2$. Figure 3.1 illustrates the regions divided by 3 lines in $\mathbb{R}^2$.

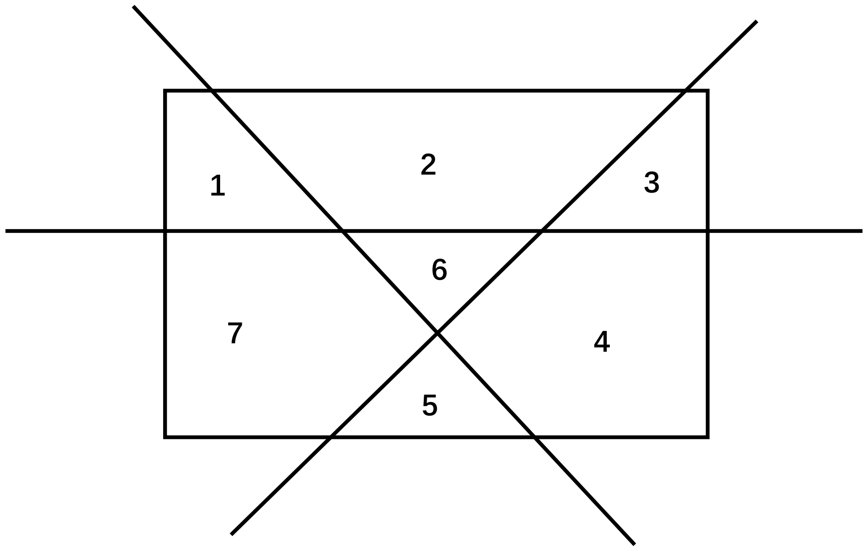


Figure 3.1: Domain Region Partitioned by 3 Lines in 2 Dimensions

For n hidden nodes with d dimensions, there should be n hyperplanes to partition the domain $\mathbb{R}^d$. The number of regions divided by hyperplanes could be expressed as

$$\begin{aligned} r(A) &= 1 + n + \binom{n}{2} + ... + \binom{n}{d} \\ &= \sum_{k=0}^{d} \binom{n}{k} \end{aligned}$$

for a real arrangement A [38]. Those divided regions are equivalent to finite elements.

Hence, it is straightforward to show that

$$r(A) \backsimeq O(n^d)$$

In other words, for a ReLU network with n hidden nodes, it could get at most n^d finite elements. However, if applying FEM, to get the same number of finite elements, it requires to approximate parameters $g^{(j,A)}$,$g^{(j,B)}$,$g^{(j,C)}$ at vertices A, B, and C in divided triangle j. The approximation function could apply basis function proposed by Bastian [37]. Thus, for FEM, h partitions are associated with h approximation models realized by basis functions, while neural networks can generate larger number of regions with the same number of hidden nodes.

It is also worth to notice that the hyperplanes of d dimensions partition the domain of R^d, instead of the function domain $\Omega\,(\Omega \subset R^d)$. In consequence, some partition hyperplanes could land in regions outside the function domain, which could decrease the approximation results. Even though such situations could be improved by providing sufficient training to adjust the weights and biases of those hyperplanes, the accuracy could be reduced under practical conditions.

3.2.1 Issues Encountered

It is difficult to get satisfying results to verify the theoretical arguments by implementing ReLU networks to approximate function solutions. This is mainly due to training networks insufficiently. The initialization of adjustable weights and biases are randomly generated, therefore, there is no guarantee the hyperplanes for hidden nodes could land within the function domain, which means those NNs may not converge. Besides, the training process could become more and more expensive as hidden nodes increase, which could cost lots of time on training. Therefore, the arguments made in Section 3.2 are remained to be verified by further experiments. The implementation details and improvement suggestions are described in Section 4.2.

Chapter 4

Implementation

4.1 Parametric Solution of ODEs

4.1.1 Testing Linear Example

The proposed trial solution solving ODEs with parametric initial condition and the cost function required to be minimized are illustrated in 3.1 and 3.2. The linear ODE example with parametric initial condition demanded to be solved is provided as follow:

$$\begin{cases} y^{'}(t) & = & y(t), \\ y(0) & = & A \end{cases} \tag{4.1}$$

whose analytic solution is $y(t) = Ae^t$.

4.1.2 Measurement Methods

Here the evaluation methods, Max norm and Euclidean norm, are performed to evaluate the NN performance by computing their relative errors. The solution for (4.1) depends on the value of A as well. Hence, it also could be written as Ψ_{nn} for the NN solution, and Ψ_a for the analytic solution.

Max Norm

For the relative error Max norm:

$$rel\ err_\infty \ := \ \frac{\|\Psi_{nn} - \Psi_a\|_\infty}{\|\Psi_a\|_\infty},$$

where the L_∞-norm for any continuous $\Phi : [x_0, x_0 + T] \times [a_0, a_0 + S] \to \mathbb{R}$ is defined as:

$$d(\Psi_{nn},\ \Psi_a) \ := \ max\{|\,\Psi_{nn}(x,\,a) - \Psi_a(x,\,a)\,|\ |\ x \in [\,x_0,\ x_0 + T\,], a \in [\,a_0,\ a_0 + S\,]\},$$

$$\|\Psi\|_\infty := \max\{|\Psi(x,\, a)| \mid x \in [\,x_0,\, x_0 + T\,], a \in [\,a_0,\, a_0 + S\,]\}.$$

Euclidean norm

For the relative error Euclidean Norm:

$$rel\ err_2 \ := \ \frac{\|\Psi_{nn} - \Psi_a\|_2}{\|\Psi_a\|_2},$$

where, Euclidean norm L_2-norm is defined as:

$$d(\Psi_{nn},\ \Psi_a) \ := \ \sqrt{\int_{x_0}^{x_0+T} \int_{a_0}^{a_0+T} |\Psi_{nn}(x,\, a) - \Psi_a(x,\, a)|^2 \, da \, dx},$$

$$\|\Psi\|_2 := \sqrt{\int_{x_0}^{x_0+T} \int_{a_0}^{a_0+T} |\Psi(x,\, a)|^2 \, da \, dx}.$$

4.1.3 Experimental Results

As for the parameter settings of this experiment, the numbers of discretization points for input X and A are both 10, and the number of hidden units is also 10. The learning rate is 0.001, the setting iteration number is 1000, and the tolerant rate for relative errors is 0.05. After optimizing parameters weights and biases for this NN, the recording results are demonstrated in Table 4.1, Figure 4.1 and Figure 4.2:

	Max Norm	Euclidean Norm
Relative Errors in Final Iteration	0.095	0.050
Training Iteration Number	1000	320
CPU Time (s)	1215.516	420.594
Wall Time (s)	1234.416	430.167

Table 4.1: Recording Results

4.2 Verifying the Arguments of How the Relative Errors Change with ReLU Networks Hidden Nodes

Based on the arguments illustrated before, the relative errors should decrease when the number of hidden nodes increases, which also satisfies the formula

$$size_{ReLU} = O(\epsilon^{-\frac{d}{m}}),\ with\ \|f - f_N\| \leq \epsilon,$$

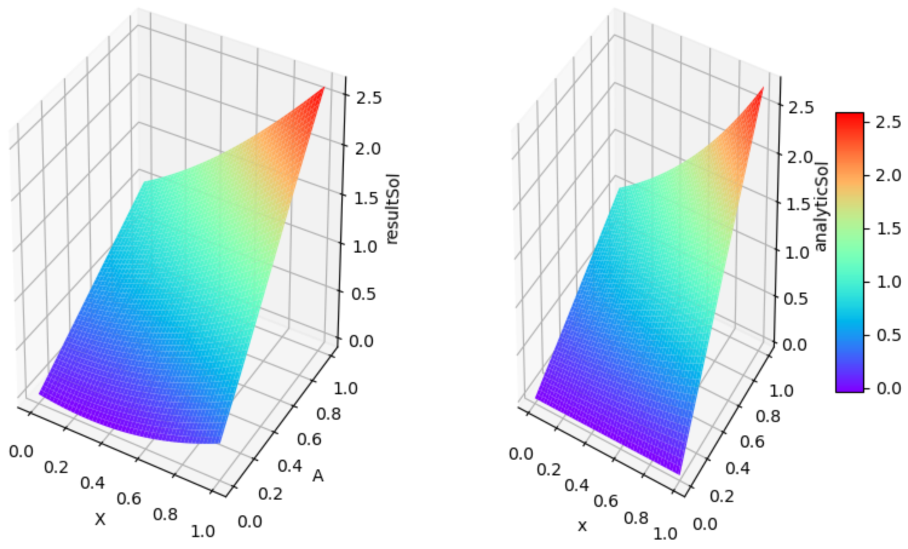
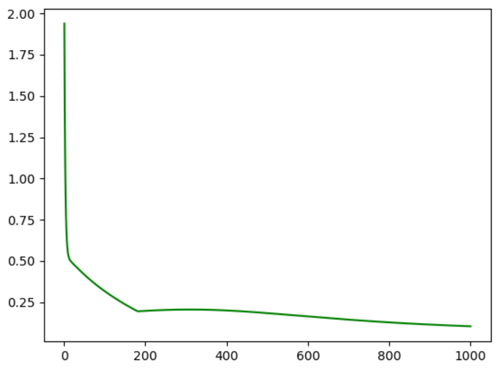
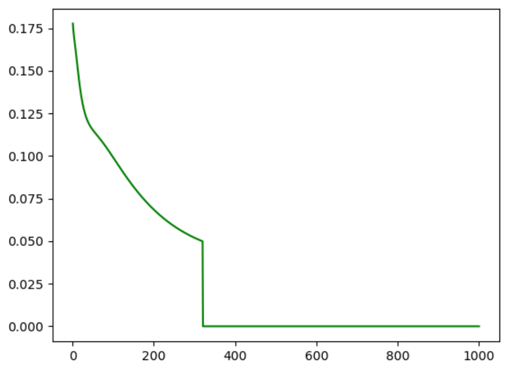


Figure 4.1: 3D Plot of Analytical (right) versus Neural Network Solution (left).



(a) Max Norm



(b) Euclidean Norm

Figure 4.2: Relative Errors in Each Iteration

where f_N is ReLU network, f is a real value function in $\mathbb{R}^d$ with input dimension d and the order of integrable m. However, the verification difficulties are hard to deal with via massive experiments. The experiments details are explained below.

The experiments are implemented by applying NumPy library in python3. Since NumPy provides lots of high-level mathematical operations and can deal with large matrix computation efficiently within relatively short time compared with other tools, such as TensorFlow. For the start of the experiments, the numerical techniques to calculate derivatives are implemented by Autograd, which is contained in python library to perform automatic differentiation for functions. Although it is convenient and transplantable to apply this tool as a general derivatives calculator, using this tool to solve partial differential equations in automatic way could get undesirable results with much longer training time. Therefore, we decide to implement derivative calculation in symbolic way by hard-coding and hand-calculation.

4.2.1 Accuracy of ReLU Networks Measured by Sobolev Norm

For this test example, we use function with two inputs ($d = 2$) within domain $\Omega = [0,\ 1] \times [0,\ 1]$

$$u(x_1,\ x_2) \ = \ 10x_1(1-x_1)x_2(1-x_2)(1-2x_2),$$

its partial derivatives with respect to two inputs could be easily calculated as

$$\frac{\partial u}{\partial x_1} = 10(1-2x_1)x_2(1-x_2)(1-2x_2),$$
$$\frac{\partial u}{\partial x_2} = 10x_1(1-x_1)(6x_2^2-6x_2+1).$$

In this test, we adopt Adam stochastic gradient decent as optimization method. As for function domain Ω, discretization points are generated by creating reticulated coordinates. We take 100 points for each input $x_1 \in [0,\ 1]$ *and* $x_2 \in [0,\ 1]$. Such that, we could get 396 discretization points set S on the boundary and 9604 points set T within the boundary.

The relative errors are calculated by using Sobolev norm as evaluation method, which could be expressed as

$$E_{Sobolev} := \frac{\|N-U\|_{1,\ 2}}{\|U\|_{1,\ 2}},$$

where N and U are the results of NNs and test function respectively. For any $\Phi : \Omega \to \mathbb{R}$, we define

$$\|\Phi\|_{1,\ 2} = \left(\int_\Omega |\Phi|^2\right)^{\frac{1}{2}} + \left(\int_\Omega |\nabla\Phi|^2\right)^{\frac{1}{2}},$$

where $\nabla\Phi$ is the gradient vector. If $\Omega \subseteq \mathbb{R}^2$, then $\nabla\Phi(x,\ y) = [\frac{\partial\Phi}{\partial x}\ \frac{\partial\Phi}{\partial y}]$. Then we could get

$$|\nabla\Phi(x,\ y)|^2 = \left|\frac{\partial\Phi}{\partial x}\right|^2 + \left|\frac{\partial\Phi}{\partial y}\right|^2.$$

It it worth to mention that for $x_i, y_i \in \Omega$, the sum of the gradient vectors' difference should be

$$|\nabla N(x_i,\ y_i) - \nabla U(x_i,\ y_i)|^2 = \left(\frac{\partial N}{\partial x_i} - \frac{\partial U}{\partial x_i}\right)^2 + \left(\frac{\partial N}{\partial y_i} - \frac{\partial U}{\partial y_i}\right)^2.$$

Besides, in our experiments, we define the derivatives of ReLU function as below

$$\frac{\partial ReLU(t)}{\partial x} = \begin{cases} 1 & if \quad t > 0 \\ 0 & if \quad t \leq 0 \end{cases},$$

at $t = 0$, we set the derivative of ReLU to 0, however, this point is a bit tricky to deal with in theory.

Thus, cost function for this teat should be

$$C_{Sobolev} = \sum_{t_i \in T} |N(t_i) - U(t_i)|^2 + \sum_{s_i \in S} |N(s_i) - U(s_i)|^2 + \sum_{t_i \in T} |\nabla N(t_i) - \nabla U(t_i)|^2,$$

where t_i, s_i are pair points (x_i, y_i) in domain T (points set within the boundary) and S (points set on the boundary) respectively. It is worth to notice that the derivatives of points on the boundary are tricky to deal with, therefore, we decide not to add them in the cost function.

After training for 3000 iteration for each preset H, the experiment results are illustrated in Table 4.2 and Figure 4.3.

H	Relative Errors
5	1.0409726555705
10	0.747550130062324
15	0.9422531059399
20	1.19008353317882
25	1.54265699

Table 4.2: Relative Errors Calculated by Sobolev Norm under Different Number of Hidden Nodes

As shown in the results, the relative errors improve instead of decreasing as the theoretical arguments predict. By applying Autograd library to calculate derivatives, we get similar experiment results. Therefore, we could exclude errors from hand-calculation for function derivatives. However, this test fails to verify the theoretical arguments.

4.2.2 Accuracy of ReLU Networks Measured by L_2-Norm

For this test example, we use function with two inputs ($d = 2$) within domain $\Omega = [0,\ 1] \times [0,\ 1]$

$$v(x_1,\ x_2) \ = \ 10x_1(1 - x_1)x_2(1 - x_2),$$

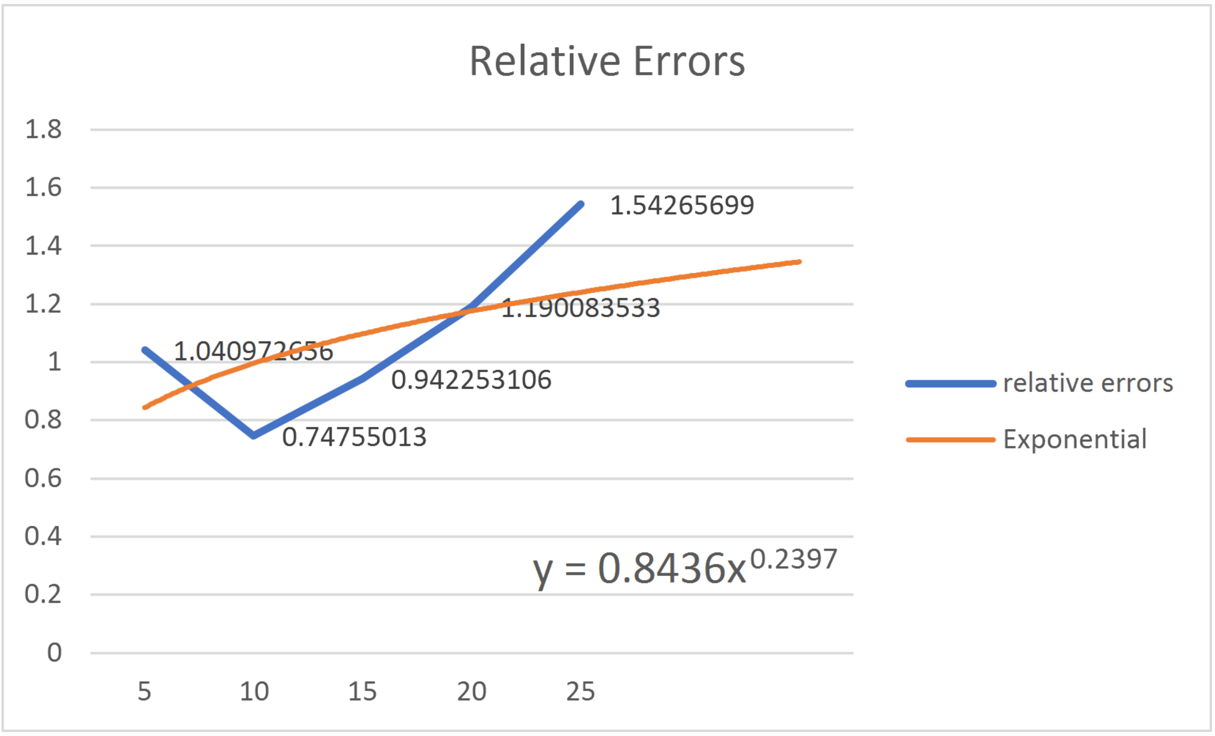


Figure 4.3: Apply Sobolev norm to estimate how relative errors (y) change with respect to hidden nodes (x). The blue line shows the experiment results. The orange line illustrates the exponential trend of the experiment results. The calculated formula demonstrates their relationship. More importantly, its exponent is 0.2397, which is far from theoretical value $-\frac{d}{m} = 2$.

its partial derivatives with respect to two inputs could be easily calculated as

$$\frac{\partial v}{\partial x_1} = (1-2x_1)x_2(1-x_2),$$
$$\frac{\partial v}{\partial x_2} = x_1(1-x_1)(1-x_2).$$

We also adopt Adam stochastic gradient decent and create mesh grid in domain Ω. 50 points are taken uniformly for each input $x_1 \in [0,\ 1] and x_2 \in [0,\ 1]$. We could get 196 discretization points set S on the boundary and 2304 points set T within the boundary.

The relative errors are calculated by using L_2-norm as evaluation method. Therefore, the cost function should set as

$$C_{L_2} = \sum_{t_i \in T} |N(t_i) - U(t_i)|^2 + \sum_{s_i \in S} |N(s_i) - U(s_i)|^2,$$

where t_i, s_i are pair points (x_i, y_i) in domain T (points set within the boundary) and S (points set on the boundary) respectively.

Furthermore, we set terminate evaluation to reduce unnecessary training iterations by monitoring the ratio of the maximum and the minimum values within 100 iterations. If the ratio is less than preset tolerance rate, the training program will be terminated before reach the maximum iteration times, 30000. Then, for each preset H, the experiment results are illustrated in Table 4.3 and Figure 4.4.

H	Relative Errors
5	0.64324449
10	0.62857409
15	0.67731311
20	0.12518816
25	0.10166721
30	0.13693964
40	0.09688802
60	0.06527643

Table 4.3: Relative Errors Calculated by L_2-Norm under Different Number of Hidden Nodes

Based on the results, the apparent decline trend is shown while increasing the number of hidden nodes. Nevertheless, the analytic outcomes of the trendline with exponentiation is not satisfying, which is calculated as $y = 0.7788x^{-1.234}$ by Excel. Since its exponent -1.234 for x is bigger than the theoretical value $-\frac{d}{m} = -2$, it fails to verify the arguments.

It is assumed reasonably that random initialization for adjustable parameters weights and biases might cause its partition hyperplane land outside the function domain $\Omega\,(\Omega \subset \mathbb{R}^d)$, which could provide high costs and relative errors for the start of the training. Although, NNs training process could help to adjust the parameters for those hyperplanes,

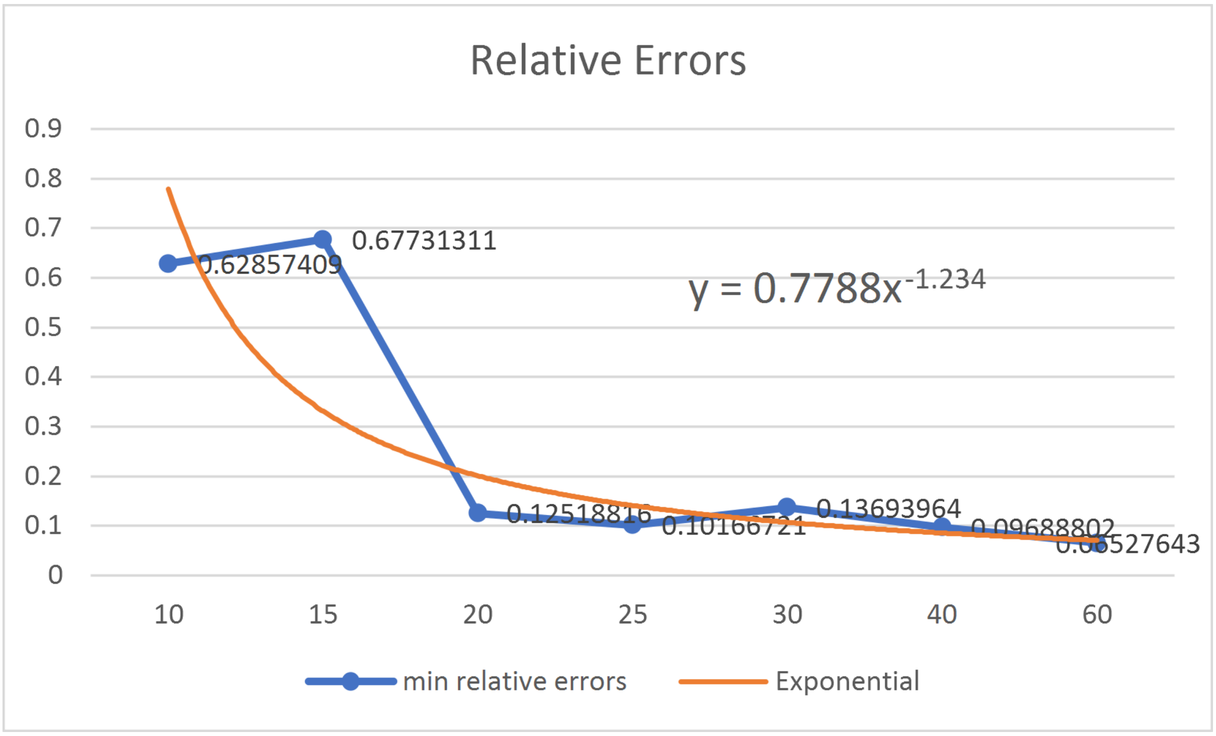


Figure 4.4: Apply L_2-norm to estimate how relative errors (y) change with respect to hidden nodes (x). The blue line shows the experiment results. The orange line illustrates the exponential trend of the experiment results. The calculated formula demonstrates their relationship. More importantly, its exponent is -1.234, which is larger than the theoretical value $-\frac{d}{m} = 2$.

the minimum relative errors as results still could be unexpected.

Hence, we update the initialization process aiming at providing fair start values for hyperplanes. Instead of initializing randomly by applying NumPy library, we assign random values for weights more intelligently. We randomly initialize weights between hidden nodes v and biases b as usual, however, the weights between inputs and hidden nodes should satisfy

$$w = -\frac{v \cdot \gamma + b}{\beta},$$

where γ and β are uniformly initialized points on the boundary $[0,\ 1]$ for 2 dimensions x_1 and x_2 respectively. By enforcing the hyperplanes to intersect with the function domain, the NNs could be converged faster than before with less iterations required. We also set 5 times to train for the same number of hidden nodes and take the minimum relative error as the representative result of this hidden node, which also helps to reduce the effect of bad initialization values randomly. Whereas we also get similar results unable to verify the theory. The experiment results are illustrated in Table 4.4 and Figure 4.5.

H	Relative Errors (experiment results)	Relative Errors (theoretical results)
2	0.53543661	0.707107
4	0.43376587	0.5
8	0.4287029	0.353553
16	0.18904418	0.25
32	0.132407840093294	0.176777
64	0.10790452	0.125
128	0.08426952	0.088388

Table 4.4: Relative Errors Calculated by L_2-Norm under Different Number of Hidden Nodes with updated initialization method

Analyzing from the results, most experiment results are less than the theoretical results for those hidden nodes. Nevertheless, the ratio of the trendline is still not desirable. The trendline could be expressed as $y = 0.7453x^{-1.013}$, whose exponent value -1.013 is still quite different from the theoretical value $-\frac{d}{m} = 2$. Thus, we are still unable to verify the arguments after lots of efforts.

4.2.3 Experiment Challenges

Those two tests are unable to verify the arguments due to the NNs training challenges for this subject. The error-tolerance rate of the ReLU networks is relatively low, as the convergency of NNs are sensitive to the function domain. Random initialization is easily to be involved in bad start values leading the hyperplanes outside the domain, which could cause networks fail to converge. Not to mention dealing with derivates could be more intractable than we expected. Therefore, finding a good training method to

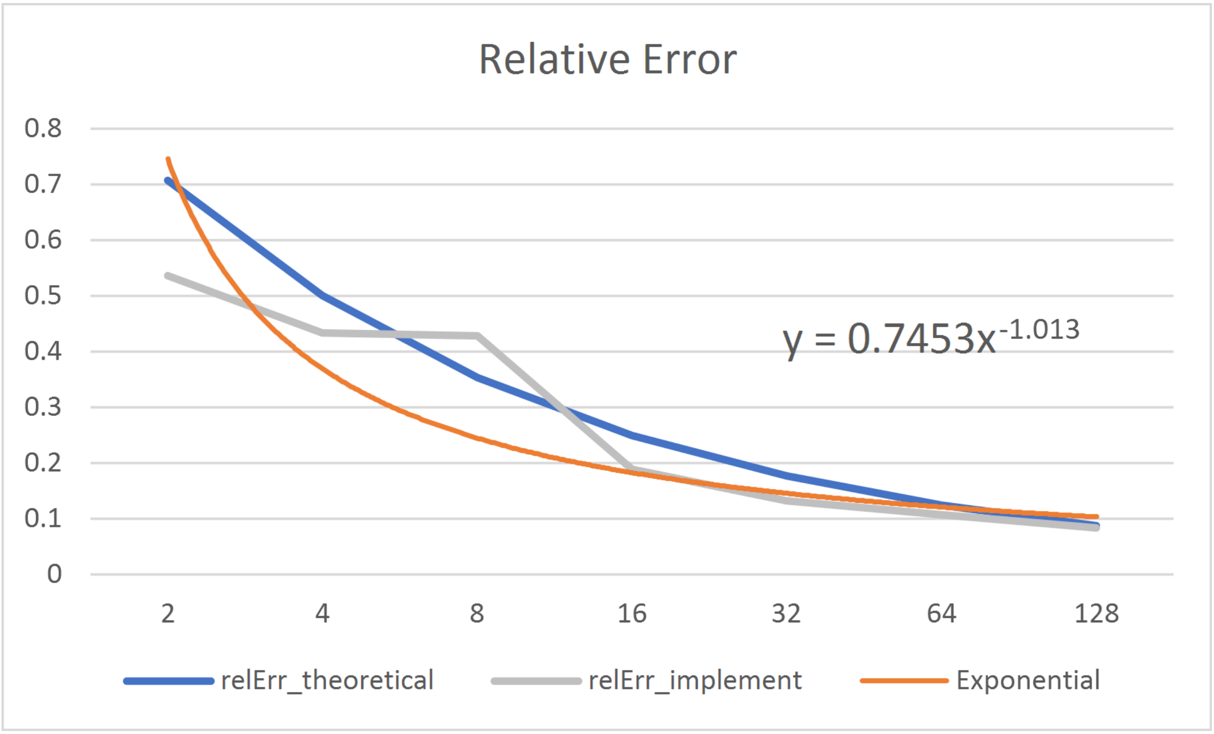


Figure 4.5: Apply L_2-norm to estimate how relative errors (y) change with respect to hidden nodes (x). The gray line shows the experiment results. The blue line demonstrates the theoretical results. And the orange line illustrates the exponential trend of the experiment results. The calculated formula demonstrates their relationship. More importantly, its exponent is -1.013, which is also larger than the theoretical value $-\frac{d}{m} = 2$.

attain low relative errors and verify the theoretical arguments for this subject are not straightforward. Possible issues related to the experiments are listed below

Possible Explanations:

1. **Insufficient Network Training**: Terminate training a bit early with relatively high tolerance rate. The repeated training iterations for the same hidden nodes are not enough, therefore, the randomly generated parameters still could make the hyperplane to land outside function domain.

2. **Initialization Issues**: No guarantee the weights between hidden nodes and biases could be initialized intelligently. There is possibility the NN could not converge.

3. **Inappropriate Optimization Method**: Apply Adam stochastic gradient decent might not be suitable enough for those tests.

4. **Theory Unverifiable**: It is presumable that the theoretical exponent values are unable to reach, since it requires quite small relative errors to verify, which could not be attained under practical conditions.

Chapter 5

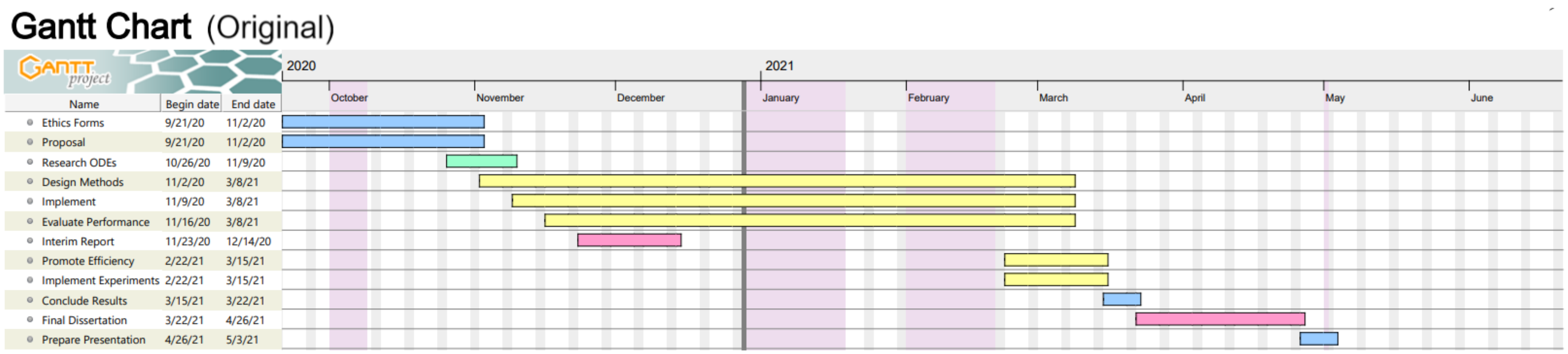
Progress

5.1 Project management

Based on the original Gantt chart and plans shown in Figure 5.1, the overall structure of this project is to start with learning solving ODEs with neural network methods and implement neural networks to solve distinguished types of ODEs. Then, this project leads to design and optimize methods of using neural networks to solve parametric ODEs. As coming up with optimized methods of solving ODEs parametrically requires lots of theoretical knowledge, research of this subject will take more significant part of this project. Implementing designed neural networks and gaining experiment results to promote the training efficiency are also important for this project.

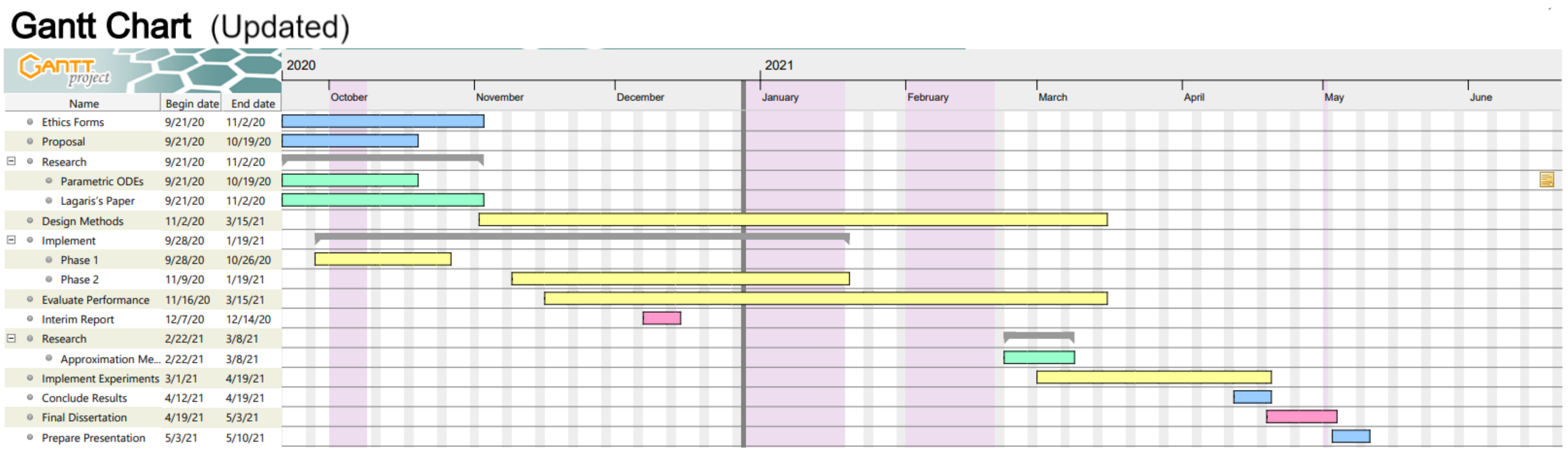
However, since this project is more about researching and proposing new theoretical methods, some adjustments of project plan are made for project works shown in Figure 5.2. This project started with enriching mathematical knowledge as basic background is required for this project, such as prevalent methods of solving ODEs, Euler's method [39]. Then, about one and half months spent on understanding and investigating Lagaris's paper, which consist of further understand of NN methods of solving ODEs and investigation of improving result solutions' accuracy by increasing the number of hidden units or discretization points. The implementations of training NNs for different types of ODE examples are also performed during this period. After that, lots of efforts were put into proposing and designing a new trail solution for solving an ODEs example with parametric initial conditions. After several weeks of theoretical design, implementations and evaluations of new NN methods were also performed to select the most appropriate approach.

In the second semester, we turned the main research direction into another NN application area, approximation ability. After comparing distinguished approximation methods and researching their theoretical arguments, we proposed an alternative proof to justify the ReLU network approximation method. However, we started to noticed that there is no experiment result could verify the theoretical argument. Therefore, we decided to design experiments to verify the argument. Around one and half months were spent to conduct experiments aiming at gaining promising results. Even though lots of adjustments of experimental methods were made, we failed to verify the theory. Then, we concluded and compared the negative results to generate some possible reasons for the experiment failure.


Gantt Chart (Original)

Name	Begin date	End date
Ethics Forms	9/21/20	11/2/20
Proposal	9/21/20	11/2/20
Research ODEs	10/26/20	11/9/20
Design Methods	11/2/20	3/8/21
Implement	11/9/20	3/8/21
Evaluate Performance	11/16/20	3/8/21
Interim Report	11/23/20	12/14/20
Promote Efficiency	2/22/21	3/15/21
Implement Experiments	2/22/21	3/15/21
Conclude Results	3/15/21	3/22/21
Final Dissertation	3/22/21	4/26/21
Prepare Presentation	4/26/21	5/3/21

Figure 5.1: Original Timetable


Gantt Chart (Updated)

Name	Begin date	End date
Ethics Forms	9/21/20	11/2/20
Proposal	9/21/20	10/19/20
Research	9/21/20	11/2/20
Parametric ODEs	9/21/20	10/19/20
Lagaris's Paper	9/21/20	11/2/20
Design Methods	11/2/20	3/15/21
Implement	9/28/20	1/19/21
Phase 1	9/28/20	10/26/20
Phase 2	11/9/20	1/19/21
Evaluate Performance	11/16/20	3/15/21
Interim Report	12/7/20	12/14/20
Research	2/22/21	3/8/21
Approximation Me...	2/22/21	3/8/21
Implement Experiments	3/1/21	4/19/21
Conclude Results	4/12/21	4/19/21
Final Dissertation	4/19/21	5/3/21
Prepare Presentation	5/3/21	5/10/21

Figure 5.2: Updated Timetable

5.2 Contributions and reflections

Due to the theoretical feature of this project, designing novel methods based on previous research and knowledge is critical for this project. Sometimes newly proposed methods might not perform as well as expected and the project might get stuck. After the first semester's effort, fair results came out related to solve ODE example with parametric initial condition and a new trail solution could be delivered. During the first semester, I further understood how the NN methods to solve differential equations and started to realize the essence of this problem that is to propose novel trail solutions and design appropriate NNs for parametric ODEs. It is challenging for students like me who obtain less mathematical backgrounds, however, I consider this as a good opportunity to enrich my knowledge and experience theoretical research.

In next semester, we investigated the approximation ability of NNs. After comparing the approximation results between traditional FEM and relatively novel NN methods with various activation functions, NN methods are more efficient requiring less model parameters. We provided alternative proof to support the theoretical argument that ReLU network obtains higher approximation capability than FEM. Then, we noticed that some experiments have already verified the approximation ability of NNs with sigmoidal activation function, however, the approximation power of ReLU network have not been verified yet. Therefore, we would like to design experiments to verify the theoretical arguments of ReLU network.

Only two-dimensional inputs were considered for test functions. We first deigned cost function and implemented NNs with Adam stochastic gradient descent as optimizers to approximate the test function and its derivatives. Symbolic calculation was implemented by hand-calculation and hard-coding. Although it was less convenient and non-portable, symbolic calculation is the most accurate compared with solving functions numerically and automatically. Then, we used Sobolev norm to evaluate the approximation accuracy. However, relative errors did not go down as we expected. They improved with the increase of hidden nodes, which is abnormal and contracts the theoretical argument, even though lots of efforts, such as adjust NN training parameters, were devoted into. Since the results are not affirmative, we only investigated approximating a test function without its derivatives evaluated by L_2-norm. Moreover, we adopt randomizing initialization intelligently. Nevertheless, the results are also negative and fail to verify the theoretical argument. The reasons why we failed to verify the argument are listed in Section 4.2.3.

5.3 Future Works

Although those experiments fail to verify the arguments currently with efforts, here are some suggestions provided for the future works of this subject:

1. **Provide Sufficient Training**: Repeat the training of the same hidden nodes for

more than 5 times. Try lower tolerance rate and preset the total iteration times to a relatively high value, such as $30,000$. Adjusting the learning rate is necessary when attempt to perform other optimization methods.

2. **Attempt Other Optimization Methods**: To some extent, Adam is considered as the best algorithm among most of the optimization methods, such as Stochastic Gradient Descent, Mini Batch Stochastic Gradient Descent, which could be stuck in local minima. However, this method might not provide the best results. Thus, other optimization methods will not land in local minima are worthy of trying.
3. **Initialize Parameters More Intelligently**: Design more intelligent initialization methods for weights and biases, such that can enforce the hyperplanes within function domain.

Bibliography

[1] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[2] Hadrien Montanelli and Qiang Du. New error bounds for deep relu networks using sparse grids. *SIAM Journal on Mathematics of Data Science*, 1(1):78–92, 2019.

[3] Alaeddin Malek and R Shekari Beidokhti. Numerical solution for high order differential equations using a hybrid neural network—optimization method. *Applied Mathematics and Computation*, 183(1):260–271, 2006.

[4] J Stoer and R Bulirsch. Topics in integration. In *Introduction to Numerical Analysis*, pages 125–166. Springer, 1993.

[5] Uri M Ascher and Linda R Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*, volume 61. Siam, 1998.

[6] Pradeep Ramuhalli, Lalita Udpa, and Satish S Udpa. Finite-element neural networks for solving differential equations. *IEEE transactions on neural networks*, 16(6):1381–1392, 2005.

[7] Hyuk Lee and In Seok Kang. Neural algorithm for solving differential equations. *Journal of Computational Physics*, 91(1):110–131, 1990.

[8] Andrew J Meade Jr and Alvaro A Fernandez. The numerical solution of linear ordinary differential equations by feedforward neural networks. *Mathematical and Computer Modelling*, 19(12):1–25, 1994.

[9] C. Lee, A. Lee, and J. Lee. Handbook of quantitative finance and risk management. pages 1–15. Springer, New York, 2010.

[10] A. N. Kolmogorov. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Doklady Akademii Nauk SSR*, 114:953–956, 1957.

[11] Alan Lapedes and Robert Farber. How neural nets work. In *Evolution, learning and cognition*, pages 331–346. World Scientific, 1988.

[12] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, 3(5):551–560, 1990.

[14] HD Vinod and Aman Ullah. Flexible production function estimation by nonparametric kernel estimators. 1987.

[15] Carlo Poloni, Andrea Giurgevich, Luka Onesti, and Valentino Pediroda. Hybridization of a multi-objective genetic algorithm, a neural network and a classical optimizer for a complex design problem in fluid dynamics. *Computer Methods in Applied Mechanics and Engineering*, 186(2-4):403–420, 2000.

[16] Yuehaw Khoo, Jianfeng Lu, and Lexing Ying. Solving parametric pde problems with artificial neural networks. *arXiv preprint arXiv:1707.03351*, 2017.

[17] Robert J Schalkoff. *Artificial neural networks*. McGraw-Hill Higher Education, 1997.

[18] PD Picton. *Introduction to neural networks*. Macmillan International Higher Education, 1994.

[19] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[20] Selwyn Piramuthu, Michael J Shaw, and James A Gentry. A classification approach using multi-layered neural networks. *Decision Support Systems*, 11(5):509–525, 1994.

[21] Richard Lippmann. An introduction to computing with neural nets. *IEEE Assp magazine*, 4(2):4–22, 1987.

[22] Alan Lapedes and Robert Farber. How neural nets work. In *Neural information processing systems*, pages 442–456, 1987.

[23] Neha Yadav, Anupam Yadav, Manoj Kumar, et al. *An introduction to neural network methods for differential equations*. Springer, 2015.

[24] Bo-An Liu and Bruno Jammes. Solving ordinary differential equations by neural networks. In *Proceeding of 13th European Simulation Multi-Conference Modelling and Simulation: A Tool for the Next Millennium, Warsaw, Poland*, volume 14, 1999.

[25] AD Polyanin and AI Zhurov. Parametrically defined differential equations. In *Journal of Physics: Conference Series*, volume 788, page 012028. IOP Publishing, 2017.

[26] Andrei D Polyanin and Alexei I Zhurov. Parametrically defined nonlinear differential equations and their solutions: Applications in fluid dynamics. *Applied Mathematics Letters*, 55:72–80, 2016.

[27] Sohrab Effati and Morteza Pakdaman. Artificial neural network approach for solving fuzzy differential equations. *Information Sciences*, 180(8):1434–1457, 2010.

[28] Ricky TQ Chen and David K Duvenaud. Neural networks with cheap differential operators. In *Advances in Neural Information Processing Systems*, pages 9961–9971, 2019.

[29] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31:6571–6583, 2018.

[30] G. G. Lorentz. The thirteenth problem of hilbert. *Proceedings of Symposia in Pure Mathematics*, 28:419–430, 1976.

[31] Igor Aleksander and H Morton. Neural computing 2. 1991.

[32] Bunpei Irie and Sei Miyake. Capabilities of three-layered perceptrons. In *ICNN*, pages 641–648, 1988.

[33] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[34] John Makhoul, Richard Schwartz, and Amro El-Jaroudi. Classification capabilities of two-layer neural nets. In *International Conference on Acoustics, Speech, and Signal Processing,*, pages 635–638. IEEE, 1989.

[35] A Ronald Gallant and Halbert White. There exists a neural network that does not make avoidable mistakes. In *ICNN*, pages 657–664, 1988.

[36] Michael I Jordan. Generic constraints on underspecified target trajectories. In *Proceedings of the International Joint Conference on Neural Networks*, volume 1, pages 217–225. IEEE Press New York, 1989.

[37] Bastian E Rapp. *Microfluidics: modeling, mechanics and mathematics*, chapter 32, pages 655–678. William Andrew, 2016.

[38] Richard P Stanley et al. An introduction to hyperplane arrangements. *Geometric combinatorics*, 13(389-496):24, 2004.

[39] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Number 44. Cambridge university press, 2009.